Lecture 8: Functions and modules

In our previous lectures, we introduced the basic tools of programming: **data types** and **control flow**.

Along with operators and variables assignments, **data types** are the building blocks of our programs and **control flow** is what allows us to organise the set of instructions in a meaningful way. Using these tools, we could potentially write complex programs already.

However, as our programs grow in size and complexity so does the set of instructions that we need to write. Chances are that some of these instructions will be repeated over and over in several parts of our code. This creates two problems:

1. The code is unnecessarily long and hard to read
2. The code is hard to debug and edit
    - Fixing an error in a section of the program that performs a specific task requires editing several blocks of code

A solution to the above problems is **Modularity**:

> **Modularity** is the concept of breaking up complex codes into smaller units that perform specific tasks.

Modularity can be thought of as the equivalent to specialisation and outsourcing in Economics.

Code modularity in Python is enabled by **Functions**:

> **Functions** are *self-contained* blocks of code that allow us to encapsulate a set of instructions related to a specific task

In this lecture, we introduce functions and modules in Python. We will start with simple examples and build on these to show all properties and usage of functions.

Functions

Suppose that you have written the following code:

```
# Block of code 1
 <statement(s)>
print("Hi there! Welcome to Python.")

# Block of code 2
<statement(s)>
print("Hi there! Welcome to Python.")

# Block of code 3
<statement(s)>
print("Hi there! Welcome to Python.")

# Block of code 4
<statement(s)>
print("Hi there! Welcome to Python.")
```

where `<statement(s)>` are blocks of code that perform operations that are potentially *different* from each other.

For whatever reason, at the end of each block of code you want the program to print a string that greets the user with a welcome message. This has been implemented above by repeating the print command over and over. Notice that since the `<statement(s)>` blocks are potentially different a **loop** could not be used in this program.

Do not Repeat Yourself (DRY): the case for functions

Imagine now that you want to change your greeting message to `"Hello there. Welcome to Python!"` . That requires

- Searching within your code for all occurences of the `print()` statement
- Manually editing all strings within the `print()` statement

This lengthy and error-prone approach can be avoided by using a **function** that performs the greeting task instead.

The greeting function can be defined as follows:

In [1]:

```
# Define a function that greets the user with a welcome message
def greeting():
    print("Hi there! Welcome to Python.")
```

The function definition above allows us to give a **name** to all statements contained *within* the **function body**, i.e. the indented block of code.

Whenever we want to the greeting task to be performed, we can **call** the greeting function by typing its name in our code.

In [2]:

```
# Block of code 1
# <statement(s)>
greeting()

# Block of code 2
# <statement(s)>
greeting()

# Block of code 3
# <statement(s)>
greeting()

# Block of code 4
# <statement(s)>
greeting()
```

```
Hi there! Welcome to Python.
Hi there! Welcome to Python.
Hi there! Welcome to Python.
Hi there! Welcome to Python.
```

The greeting message can now be changed very easily by simply changing the body of our function.

In [3]:

```
# Update the greeting message by editing the body of the function
def greeting():
    print("Hello there. Welcome to Python!")
```

The above program can now be run **without any modification to any of its lines of codes**. The greeting message, however, will be updated.

In [4]:

```
# Block of code 1
# <statement(s)>
greeting()

# Block of code 2
# <statement(s)>
greeting()

# Block of code 3
# <statement(s)>
greeting()

# Block of code 4
# <statement(s)>
greeting()
```

```
Hello there. Welcome to Python!
Hello there. Welcome to Python!
Hello there. Welcome to Python!
Hello there. Welcome to Python!
```

Passing data to functions: parameters and arguments

Functions can do much more than simply giving a name to a set of instructions. For instance, functions can use information or data generated in other parts of the code or provided by the user to perform a set of instructions that **depend on that data**.

Let's say that we want to personalise the greeting message above to display the name of the user instead of a generic greeting message. We can do that by **passing** that the name into the function when this is called. Internally, the function will use the name to print a personalised message.

In [5]:

```
# Update the greeting function to take a name as an argument
def greeting(name):
    print(f"Hi {name}! Welcome to Python.")
```

We can now personalise the greetings in the original program by passing different strings contaning the user names as **arguments** in the function call.

In [6]:

```
# Block of code 1
# <statement(s)>
greeting("Luca")

# Block of code 2
# <statement(s)>
greeting("Roger")

# Block of code 3
# <statement(s)>
greeting("Rafael")

# Block of code 4
# <statement(s)>
greeting("Novak")
```

```
Hi Luca! Welcome to Python.
Hi Roger! Welcome to Python.
Hi Rafael! Welcome to Python.
Hi Novak! Welcome to Python.
```

Notice the difference between the **function definition**

```
def greeting(name):
    print(f"Hi {name}! Welcome to Python.")
```

and the **function call**

```
greeting("Luca")
```

In a function definition context, the `name` keyword is known as a **parameter**. This keyword is chosen once when you define the function and fixed throughout.

In a function call context, the string typed within the parenthesis `()` is known as the **argument**. Arguments can change at each call of the function and carry the information needed by the function.

When the function is called, the value of the argument `"Luca"` is **bound** to the keyword `name`. It is **as if** a variable `name` with value `"Luca"` was defined within the body of the function.

When the statements within the function body are executed, Python will substitute the string `"Luca"` wherever the keyword `name` is used.

Multiple parameters and arguments

Functions can take many arguments as inputs.

Let's say that we want to enrich our message by displaying the user surname and the version of Python they use. This is achieved by adding two parameters to our `greeting()` function definition: `surname` and `version`.

In [7]:

```
# Update the greeting function to take multiple arguments as inputs
def greeting(name, surname, version):
    print(f"Hi {name} {surname}! Welcome to Python {version}.")
```

Once a function with multiple parameters is defined, we need a systematic way to pass the correct information to the relevant parameter when we call the function.

There are two ways of passing arguments to a function:

- **Positional arguments**
- **Keyword arguments**

Positional arguments

Arguments are passed into the function call as an **comma-separated list** of values, variable names, or expressions.

Arguments **must be listed in the same order as that of parameters** in the function definition.

In [8]:

```
# Correct call via direct passing of values
greeting("Luca", "Rondina", 3.9)

# Correct call via passing of variable names
roger_version = 3.8
greeting("Roger", "Federer", roger_version)

# Correct call via passing of variable names and expressions
rafa_surname = "Nadal"
greeting("Rafael", rafa_surname, roger_version - 1)

# WRONG CALL due to INCORRECT ORDER
greeting(2, "Novak", "Djokovic")
```

```
Hi Luca Rondina! Welcome to Python 3.9.
Hi Roger Federer! Welcome to Python 3.8.
Hi Rafael Nadal! Welcome to Python 2.8.
Hi 2 Novak! Welcome to Python Djokovic.
```

Keyword arguments

Arguments are passed into the function call as a comma-separated list of `keyword=value` pairs.

The order of the arguments is not relevant but the `keyword` s **must match the parameter name** in the function definition.

Notice the arguments order and `keywords` in the example below:

In [9]:

```
# Correct call via direct passing of values
greeting(version=3.9, name="Luca", surname="Rondina")

# Correct call via passing of variable names
roger_version = 3.8
greeting(surname="Federer", name="Roger", version=roger_version)

# Correct call via passing of variable names and expressions
rafa_surname = "Nadal"
greeting(name="Rafael", version=roger_version - 1, surname=rafa_surname)

# WRONG CALL due to INCORRECT KEYWORD
greeting(name="Novak", surname="Djokovic", python_version=2)
```

```
Hi Luca Rondina! Welcome to Python 3.9.
Hi Roger Federer! Welcome to Python 3.8.
Hi Rafael Nadal! Welcome to Python 2.8.

---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-9-b98356b9602f> in <module>
     11
     12 # WRONG CALL due to INCORRECT KEYWORD
---> 13 greeting(name="Novak", surname="Djokovic", python_version=2)

TypeError: greeting() got an unexpected keyword argument 'python_ver
sion'
```

Positional and keyword arguments can be combined in a function call. When doing so all positional arguments must be specified **before** keyword arguments.

In [10]:

```
# Correct order of positional and keyword arguments
greeting("Luca", "Rondina", version=3.9)

# WRONG order of positional and keyword arguments
greeting("Novak", surname="Djokovic", 2)
```

```
  File "<ipython-input-10-4f1410ecf57f>", line 5
    greeting("Novak", surname="Djokovic", 2)
                                          ^
SyntaxError: positional argument follows keyword argument
```

Default values

There are situations in which you may want to assign a **default value** to one or more of the parameters. When a parameter has been assigned a default value in the function definition, the associated argument becomes **optional**.

For instance, in our example we might want to specify a default Python version in case the user does not know the version they use.

In [11]:

```
# Update the greeting function to specify a default Python version
def greeting(name, surname, version=3):
    print(f"Hi {name} {surname}! Welcome to Python {version}.")
```

*Note: When using default values, any parameter with a default value needs to be listed after all the parameters without default values. This way Python understands which arguments are **required** and which are **optional**.*

In [12]:

```
# When an optional argument is NOT specified, the function falls back to the def
ault value of the parameter
greeting("Luca", "Rondina")

# When an optional argument IS specified, the function overrides the default val
ue of the parameter
greeting("Luca", "Rondina", 3.9)
```

```
Hi Luca Rondina! Welcome to Python 3.
Hi Luca Rondina! Welcome to Python 3.9.
```

Getting back data from functions: the `return` statement

As mentioned in the beginning, the main motivation behind using functions is that they allow us to simplify our program by sub-dividing it into smaller dedicated tasks.

In the examples above, we have seen how we can simplify a printing command by collecting all relevant information and passing it as input into a function.

In many situations, however, we are not interested in merely *displaying* the outcome of a sequence of operations, but rather we are interested in the outcome *itself*. This is especially true if we want to use that outcome later on in our code to perform other tasks.

As an example, let's say that we want to compute the average BMI index of a group of people given data on their height in feet and inches, and weight in pounds.

We will have to write a program that for each person

1. Takes the height as input in feet and inches, and converts it to metres
2. Takes the weight as input in pounds and converts it to kilograms
3. Use the relevant formula to compute the BMI index

and finally collects all BMI indexes to compute the average BMI index.

The key insight of **modularity** is that from the perspective of the main program, i.e. computing the average BMI, we should not be concerned with any of the intermediate steps. All we need is the BMI index of each person given their height and weight.

Steps 1 to 3 should then be considered as a "black box" that takes inputs (height, weight) and **returns** an output, the BMI index.

In Python, functions can be seen as "black boxes" that take inputs, i.e. the arguments, and give back the required data as output. Data is sent back to the main program using a `return` statement.

In [13]:

```
# Define a function that computes the BMI index
def bmi(height, weight):
    # 1. Convert the height in feet and inches to metres
    # The parameter height is a list that stores [feet, inches]
    height_m = 0.3048 * (height[0] + height[1]/12)

    # 2. Convert the weight in pounds to kg
    weight_kg = 0.453592 * weight

    # 3. Compute the BMI index using the formula
    bmi_index = weight_kg / height_m ** 2

    # Return the BMI index with a return statement
    return bmi_index
```

We can now write our main program as follows

In [14]:

```
# Compute the BMI index for each person using the bmi() function
Luca_bmi   = bmi(height=[5, 9], weight=154.324)
Marco_bmi  = bmi([5, 6], 143.3)
Andrea_bmi = bmi([6, 0], 202.825)

# Compute and display the average bmi
average_bmi = (Luca_bmi + Marco_bmi + Andrea_bmi) / 3
print(average_bmi)
```

```
24.4753659543222
```

Returning multiple values

Let's say we are interested in returning the converted height and weight to the main program in addition to the BMI index.

In Python, functions can return multiples values by specifying a comma-separated list of variables or expressions in the `return` statement.

In [15]:

```
# Modify the bmi() function to return multiple values
def bmi(height, weight):
    height_m = 0.3048 * (height[0] + height[1]/12)
    weight_kg = 0.453592 * weight
    bmi_index = weight_kg / height_m ** 2

    # Return a comma-separated list of variables or expressions
    return bmi_index, height_m, weight_kg
```

If we look closely, any comma-separated list of values in Python is always a **tuple**.

Indeed, whenever the `bmi()` function is called it is actually returning a **tuple** which can be then unpacked to assign the return values to single variables

In [16]:

```
Luca_bmi, Luca_height_m, Luca_weight_kg = bmi(height=[5, 9], weight=154.324)

print(Luca_bmi)
print(Luca_height_m)
print(Luca_weight_kg)
```

```
22.789418463188948
1.7526000000000002
70.000131808
```

Modules

We have seen how to break down a long and complex code into smaller self-contained tasks with functions. Nevertheless, we defined all of our functions into the main program itself. If we have hundreds of function definitions, the main program can become lengthy and hard to read.

A further step towards modularity is to "hide" all function definitions into **modules**. A **module** is a Python file with extension `.py` that allow us to store all of our function definitions in one place, and later reuse them in the main program.

Importing modules

In order to use the functions defined in a module in our main program we first need to import the module.

Let's say that we have a module named `body_mass_index.py` containing three functions:

- A function `bmi()` that computes the body mass index
- A function `average_bmi()` that computes the average bmi
- A function `show_bmi()` that computes and displays the bmi index

This module can be imported into the main program with the following statement

In [17]:

```
import body_mass_index
```

When invoked, the `import` statement creates a **namespace** that allows us to call the functions defined in `body_mass_index.py` .

We can call the imported functions by typing `module_name.function_name()` . For instance

In [18]:

```
# Call the bmi() function from the body_mass_index module
Luca_bmi   = body_mass_index.bmi(height=[5, 9], weight=154.324)
Marco_bmi  = body_mass_index.bmi([5, 6], 143.3)
Andrea_bmi = body_mass_index.bmi([6, 0], 202.825)

# Call the average_bmi() function from the body_mass_index module
mean_bmi = body_mass_index.average_bmi(Luca_bmi, Marco_bmi, Andrea_bmi)
print(f"The average BMI index is {mean_bmi}")

# Call the show_bmi() function from the body_mass_index module
body_mass_index.show_bmi(height=[5, 9], weight=154.324)
```

```
The average BMI index is 24.4753659543222
The BMI index is 22.789418463188948
```